TEMA V: NOCIONES DE COMPLEJIDAD COMPUTACIONAL

OBJETIVOS GENERALES

1. **Conocer el concepto de algoritmo.**
2. **Complejidad de algoritmos.**

OBJETIVOS ESPECÍFICOS

- ✓ **Conocer el concepto de algoritmo.**
- ✓ **Reconocer algunos tipos de algoritmos.**
- ✓ **Conocer el significado de las funciones O, Omega y Zeta.**
- ✓ **Complejidad algoritmica**
- ✓ **Problemas N yNP**

BIBLIOGRAFÍA

- "Matemática Discreta y sus aplicaciones". K. H. Rosen. McGraw-Hill, 2004.
- "Matemáticas discretas", R. Johnsonbaugh. Ed. Pearson Prentice Hall, 1999.
- "Algoritmos. Teoría y práctica", Gilles Brassard y Paul Bratley. Ed. Pearson Prentice Hall, 1988.
- "Matemática discreta", Norman L. Biggs. Oxford University Press, 2002.

DESARROLLO TEÓRICO

IV.1 Introducción.
IV.2 Algoritmos.
IV.3 Crecimiento de funciones.
IV.4 Complejidad algorítmica.

1. INTRODUCCIÓN

Nos centramos en problemas que se pueden resolver mediante el uso de métodos paso a paso, más formalmente conocido como **algoritmos**. Estos problemas se pueden estudiar a través de la Matemática Discreta, ya que para su resolución se necesitan conjuntos finitos o, al menos, conjuntos contables infinitos de pasos. Las computadoras son particularmente útiles aquí, porque pueden ser programadas para seguir una secuencia de instrucciones, y lo hacen rápidamente y sin cometer ningún error.

Originalmente, la palabra "algoritmo" se usaba para designar métodos usados en aritmética elemental, como sumas y productos grandes. Si un niño que sabe el procedimiento para sumar, cuando le damos dos enteros positivos, puede calcular su suma. Aunque el niño no tiene que entender el fundamento lógico del procedimiento, todo lo que se requiere es la capacidad de llevar a cabo los pasos correctos en el orden correcto.

Muchos problemas pueden resolverse considerándolos como casos especiales de un problema general. Por ejemplo, el problema de localizar el mayor entero en la lista {23, 45, 18, 53, 19}, es un caso particular del problema de localizar el mayor entero en una sucesión de enteros. Para resolver este problema general, necesitamos un **algoritmo**, que especifique una secuencia de pasos que se usan para resolver el problema general.

Una consideración importante en relación con un algoritmo es su **complejidad computacional**. Esto es, ¿qué recursos computacionales se necesitan para usar este algoritmo en la resolución de un problema de un tamaño específico? Para medir la complejidad de un algoritmo utilizaremos las notaciones O y Θ.

2. ALGORITMOS

En Matemática Discreta aparecen muchas clases de problemas genéricos. Por ejemplo: dada una secuencia de enteros, encontrar el más grande, listar todos sus subconjuntos, ponerlos en orden creciente; dada una red de comunicaciones de datos, encontrar el camino más corto entre dos ordenadores. Cuando se nos presentan tales problemas, lo primero que tenemos que hacer es construir un modelo que traduzca el problema a un contexto matemático.

Sin embargo, fijar este modelo matemático sólo es parte de la solución, para completarla se necesita un procedimiento que sigua una secuencia de pasos que conduzca a la respuesta deseada. Tal secuencia de pasos se le llama **algoritmo**. Por tanto,

Un **algoritmo** es un conjunto finito de instrucciones precisas que resuelven un problema o realizan un cálculo.

Ya conocemos algunos ejemplo en esta asignatura: algoritmo de la división, algoritmo de Euclides, algoritmo chino del resto,…; y otros que se han usado en clase de prácticas: calculo de una tabla de verdad, producto cartesiano, partes de un conjunto, retículos, algebras de Boole,…

No obstante, recurriremos en este capítulo a otros algoritmos tipo bien conocidos por el alumnado de Ingeniería Informática, como encontrar el mayor o menor elemento de una lista finita, búsqueda de un elemento en una lista finita de elementos (búsqueda lineal y búsqueda binaria), ordenación de una lista finita de elementos (la ordenación de burbuja y ordenación por inserción), etc.

El término algoritmo es una degeneración del nombre del astrólogo y matemático del siglo IX Mohamed Ben Musa Al-Jowarizmi, cuyo libro sobre numerales hindúes “Algoritmi de Numero Indorum” es la base de la notación decimal que hoy en día usamos. Originalmente, una variante de la palabra algoritmo se uso para definir las reglas usadas para hacer aritmética usando notación decimal. El término evolucionó a algoritmo en el siglo XVIII. Con el creciente interés por todo lo relativo a la computación, este concepto adquirió un significado más genérico, pasando a incluir procedimientos utilizados para resolver problemas.

Como curiosidad, los europeos aprendieron álgebra por primera vez a través de sus textos. La palabra álgebra proviene de vocablo al-jabr, parte del título del libro “Hisab al-jabr w’al muquabala”.

Se piensa que el fue el primero que usó el cero como dígito.

Abu Ja’Far Mohammed Ibn Musa Al-khowarizmi (780-850). Su nombre al-Khowarizmi significa “del pueblo de Khowarizmi” que actualmente se llama Khiva y es parte de Uzbekistan.

Podemos especificar un procedimiento para resolver un problema de varias maneras. Un método es simplemente usar el castellano para describir la secuencia de pasos usados.

Un algoritmo también se puede describir utilizando un lenguaje de programación. Sin embargo, en este caso, sólo se pueden utilizar las instrucciones permitidas en el lenguaje en cuestión. Esto a menudo conduce a una descripción del algoritmo que es complicada y difícil de entender. Teniendo en cuenta que hay muchos lenguajes de programación, en lugar de utilizar uno particular para especificar algoritmos, es común usar una forma de **pseudocódigo**, es decir, un paso intermedio entre una descripción en castellano y una implementación de este algoritmo en un lenguaje de programación. Nosotros usaremos el lenguaje de programación de Mathematica que hemos aprendido en prácticas.

Hay varias propiedades que generalmente comparten los algoritmos:

- *Entrada:* Un algoritmo tiene valores de entrada
- *Salida:* A partir de cada conjunto de valores de entrada un algoritmo produce valores de salida (la solución del problema).
- *Definición:* Los pasos de un algoritmo deben definirse con precisión.
- *Corrección:* Un algoritmo debe producir la salida correcta para cada conjunto de valores de entrada.
- *Finitud:* Un algoritmo debe producir la salida deseada después de un número finito (pero quizás grande) de pasos para cualquier entrada de datos.
- *Efectividad:* Cada paso se debe poder realizar de forma exacta y en tiempo finito.
- *Generalidad:* Puede ser aplicable a todos los problemas similares, no sólo para unos datos particulares.

Ejemplo 1. Encontrar el mayor o menor elemento, en un conjunto o sucesión de elementos finitos para un cierto orden en éste: elegimos uno cualquiera como el mayor (o menor) provisional, por ejemplo el primero, lo comparamos con todos uno a uno y cambiamos el provisional siempre que este cambie, al terminar, el provisional será mayor (o menor) definitivo.

En Mathematica:

```
In[]:= Conjunto={3, 6, 2, 1, 0, 7, 8, 5};
       Provisional=Conjunto[[1]];
       Do[
         If[Provisional<Conjunto[[i]],Provisional=Conjunto[[i]]];
       ,{i,2,Length[Conjunto]}];
       Print[“El mayor elemento es: ”, Provisional];
```

El problema de localizar un elemento de una lista ordenada se puede encontrar en muchos contextos. Los problemas de este tipo se llaman **problemas de búsqueda** y se puede describir como sigue: localizar un elemento x en una lista de elementos distintos a_1, a_2,…, a_n o determinar que no está en la lista. La solución es localizar el término x en la lista (esto es, la solución es i si $x = a_i$) o es 0 si x no está en la lista.

Ejemplo 2. (Búsqueda lineal o **busqueda secuencial)**: comienza por comparar x y a_1, cuando $x = a_1$ la solución es el índice de de a_i, es decir, 1, en otro caso x se compara con el siguiente elemento y así sucesivamente con cada término de la lista hasta que se encuentra una coincidencia. Si tras recorrer toda la lista no se localiza x, la solución es 0.

En Mathematica:

```
In[]:= Elemento=0;
       Conjunto={3, 6, 2, 1, 0, 7, 8, 5};
       i=1;
       While[Elemento!=Conjunto[[i]] && i!=Length[Conjunto],
             i++;
        ];
       If[Elemento==Conjunto[[i]],
          Print["El elemento está en la posición: ", i];
          ,Print["El elemento no está"];
        ];
```

Matemática Discreta
García Muñoz, M.A.

Ejemplo 3: (Búsqueda binaria) sólo es válido para conjunto finitos donde suponemos ordenados los elementos por alguna relación de orden total (por ejemplo, si los términos son números, estos están listados del menor al mayor; si son palabras, estas están listadas en orden lexicográfico o orden alfabético): Comienza comparando el elemento con el elemento que aparecen justo en mitad de la lista. A continuación la lista se divide en dos subconjuntos con igual, si es posible, número de elementos (los pequeños y los grandes) y comparamos con el mayor de los pequeños y el menor de los grandes. Después de deducir en que subconjunto podría estar, repetimos el proceso restringiendo la búsqueda al subconjunto apropiado.

En Mathematica:

```
In[]:= Elemento=7;
       Conjunto={1,2,3,4,5,6,7,8,9,10};
       Principio=1;
       Final=Length[Conjunto];
       While[Principio≠Final,
             Mitad=Quotient[Principio+Final,2];
             If[Conjunto[[Mitad]]>=Elemento,
               Final=Mitad, Principio=Mitad+1
              ];
       ];
       If[Elemento==Conjunto[[Principio]],
           Print[“El elemento está en la posición: ”, Principio];,
           Print[“El elemento no está”];
       ];
```

Pasos:

(1) Principio=1, Final=10, Mitad=5
{1,2,3,4,5} {6,7,8,9,10}
(2) Principio=6, Final=10, Mitad=8
{6,7,8} {9,10}
(3) Principio=6, Final=8, Mitad=7
{6,7} {8}
(4) Principio=6, Final=7, Mitad=6
{6} {7}
(5) Principio=7, Final=7
Se compara “Elemento” y 7

3. CRECIMIENTO DE FUNCIONES

Para resolver un mismo problema puede que encontremos varios algoritmos. Nos interesará determinar cuáles son los mejores (más efectivos) en términos del tiempo o memoria empleadas para resolver el problema. El tiempo necesario para resolver un problema no sólo depende del número de operaciones que realiza. También depende del hardware y del software que usemos para ejecutar el programa que implementa el algoritmo. Sin embargo, cuando cambiamos el hardware y el software usado para implementar un algoritmo, podemos aproximar el tiempo requerido para resolver un problema de tamaño conocido sólo multiplicando el tiempo empleado anteriormente por una constante.

Así tendremos que realizar este análisis con independencia del hardware o software donde se implementen. Por ello nos interesa conocer cómo se comportan dichos algoritmos según la cantidad de datos que manejen, cuántas instrucciones tiene que ejecutar el ordenador o cuánta memoria precisa, tanto en el mejor de los casos, como en el peor.

Este análisis lo realizaremos buscando una función que para un tamaño concreto (cantidad de datos) del problema nos dé el número de instrucciones a ejecutar o la memoria usada, y después también estudiaremos cómo se comporta y cómo crece dependiendo del tamaño del problema. Una de las ventajas de usar la función O, es que podemos estimar el crecimiento de una función sin preocuparnos por multiplicadores constantes o términos de orden más pequeños. Es decir, usando la notación O, no tenemos que preocuparnos por el hardware y el software utilizados.

La notación O se utiliza para estimar el número de operaciones que un algoritmo utiliza a medida que crece el tamaño de la entrada. Con la ayuda de esta notación, podemos determinar si es práctico utilizar un algoritmo para resolver un problema si se aumenta el tamaño de la entrada. Además, nos permite comparar dos algoritmos para determinar cuál es más eficiente a medida que el tamaño de la entrada crece.

Dadas dos funciones f y g de $\mathbb{R}$ o $\mathbb{Z}$ en $\mathbb{R}$, diremos que **f(x) es O(g(x))** ("f(x) es O mayúscula de g(x)") si existen dos constantes C y k tales que

$$|f(x)| \leq C|g(x)|, \quad \forall\ x>k.$$

A C y k los llamaremos **testigos** de la relación f(x) es O(g(x)) y escribiremos: f(x) = O(g(x)).

Nótese que cuando existe un par de testigos para la relación f(x) = O(g(x)), existen infinitos pares de testigos.

Una forma útil para encontrar un par de testigos is primero seleccionar un valor para k para el cual el tamaño de |f(x)| se pueda estimar cuando x > k y ver si podemos usar esta estimación para encontrar un valor de C para el cual |f(x)| < C|g(x)| para x > k.

Ejercicio 1: Mostrar que f(x) es O(g(x)) donde $f(x) = x^2 + 2x + 1$ y $g(x) = x^2$.
¿Es $g(x) = O(x^2 + 2x + 1)$?

Ambas funciones tienen **el mismo orden**

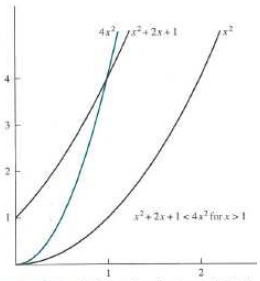

FIGURE 1 The Function $x^2 + 2x + 1$ is $O(x^2)$.

Aunque, denotamos f(x) = O(g(x)), el signo igual en esta notación no representa una igualdad. Más bien, esta notación nos dice que existe una desigualdad que relaciona los valores de las funciones f y g para números suficientemente grandes en los dominios de estas funciones.

Cuando f(x) es O(g(x)) y h(x) es una función que tiene mayor valor absoluto que g(x) para valores suficientemente grandes de x, se tiene que f(x) es O(h(x)).

Al usar la notación O, la función g en la relación f(x) es O(g(x)) es elegida de forma que sea lo más pequeña posible (algunas veces de un conjunto de referencia de funciones, tales como, las funciones de la forma x^n, donde n es un entero positivo).

Ejercicio 2: Mostrar que $5x^2$ es $O(x^3)$. ¿Es también cierto que x^3 es $O(5x^2)$?

La notación O se ha utilizado en matemáticas desde hace más de un siglo. En informática se usa en el análisis de algoritmos. Fue introducido por el matemático alemán Paul Bachmann en 1892. La notación O se le llama **símbolo de Landau** por el matemático alemán Edmund Landau, que la utilizó en sus trabajos de investigación. El uso de la notación O en informática fue popularizado por Donald Knuth, que también introdujo las notaciones Omega **Ω** y Theta Θ en los años 70.
La notación O tiene limitaciones. En particular, cuando f(x) es O (g(x)), tenemos un límite superior (en términos de g(x)) para el tamaño de f(x) para valores grandes de x. Sin embargo, la notación O no proporciona un límite inferior para el tamaño de f(x) para valores grandes de la variable. Para ello, usamos la **notación de Omega**. Cuando queremos dar tanto un límite superior como un límite inferior sobre el tamaño de una función f(x), en relación con una función de referencia g(x), utilizamos la **notación Theta**.

Dadas dos funciones f y g de $\mathbb{R}$ o $\mathbb{Z}$ en $\mathbb{R}$, diremos que **f(x) es Ω(g(x))** (f(x) es Omega mayúscula de g(x))si existen dos constantes C y k tales que

$$|f(x)| \geq C|g(x)|, \quad \forall\ x>k.$$

Escribiremos: $f(x) = \Omega(g(x))$.

Hay una conexión fuerte entre las notaciones O y Ω. Es obvio que si $C \neq 0$, entonces $g(x) = O(f(x)) \Leftrightarrow f(x) = \Omega(g(x))$.

Ejercicio 3: Mostrar que $3x^3 + x^2 + 2$ es $\Omega(x^3)$. ¿Es cierto que x^3 es $\Omega(5x^2)$?

Conocer el orden de crecimiento de una función requiere que tengamos tanto un límite superior como un límite inferior para el tamaño de la función. Por lo tanto, dada una función f(x), queremos encontrar otra función de referencia g(x) tal que f(x) = O(g(x)) y f(x) = Ω(g(x)):

Dadas dos funciones f y g de $\mathbb{R}$ o $\mathbb{Z}$ en $\mathbb{R}$, diremos que **f(x) es Θ(g(x))** si f(x) es O(g(x)) y Ω(g(x)). En tal caso diremos que "f(x) es theta mayúscula de g(x)" o que f(x) es del orden de g(x).

Cuando f(x) es Θ(g(x)), también se tiene que g(x) es Θ(f(x)).

Es fácil probar que f(x) es Θ(g(x)) si podemos encontrar dos números reales C_1 y C_2 y un número positivo k tales que:

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|, \quad \forall\ x>k.$$

Ejercicio 4: Demostrar que $3x^3 + x^2 \log x$ es $\Theta(x^3)$.

Habitualmente cuando decimos que una función f(x) es Θ(g(x)), la función g(x) es simple: 1, log(x), x, x.log(x), x^n, a^x, x!.

Usando el cálculo se puede demostrar que cada función en la lista es más pequeña que la función siguiente, en el sentido de que la relación de una función y la función siguiente tiende a cero cuando x crece sin límite.

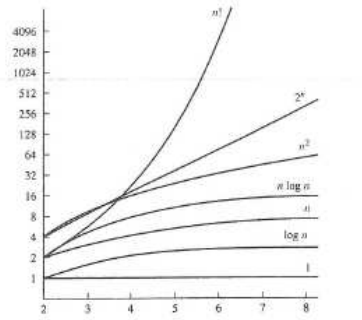

Figura 3. Representación del crecimiento de funciones utilizadas comúnmente en las estimaciones con la notación *O*.

Proposición 5.1. Las siguientes afirmaciones equivalen:

1. f(x) es de orden g(x).
2. f(x) = O(g(x)) y g(x) = O(f(x)).
3. f(x) = Ω(g(x)) y g(x) = Ω(f(x)).

Proposición 5.2. Si f(x) = O(g(x)) y g(x) = O(h(x)) entonces f(x) = O(h(x)). (Idem para Ω y Θ).

Proposición 5.3. Si $f(x) = a_n x^n + \ldots + a_1 x + a_0$ (polinomio con coeficientes reales), entonces $f(x) = O(x^n)$ (el término líder de un polinomio determina su orden).

Muchos algoritmos están formados por dos o más subprocesos separados. El número de pasos realizados por una computadora para resolver un problema con entrada de un tamaño específico con dichos algoritmos es la suma del número de pasos dado por cada subproceso. Así que la estimación en notación O de estos algoritmos dependerá de la que tenga cada subproceso y vendrá dada por la combinaciones de las estimaciones de los subprocesos:

Proposición 5.4. Supongamos que f_1 (x) es $O(g_1(x))$ y que $f_2(x)$ es $O(g_2(x))$. Entonces:

i) $(f_1 + f_{2.})(x)$ es $O(\text{máx}\{|g_1(x)|, |g_2(x)|\})$.
ii) $(f_1 f_2)(x)$ es $O(g_1(x)\, g_2(x))$.

Corolario 5.5. Si $f_1(x)$ y $f_2(x)$ son ambas $O(g(x))$ entonces
$(f_1 + f_2)(x) = O(g(x))$.

Ejercicio 5: Dar una estimación O para las funciones:
a) $f(n) = n\log(n!) + (n^2 + 1)\log n$ donde n es un entero positivo.
b) $h(x) = (x+4)\log(x^2 + x) + 5x^2$.

4. COMPLEJIDAD ALGORÍTMICA

Matemática Discreta
García Muñoz, M.A.

¿Cuándo un algoritmo soluciona de forma satisfactoria un problema? Primero, debe de darnos siempre la respuesta correcta y en segundo lugar, debería ser eficiente. ¿Cómo podemos analizar la eficiencia de los algoritmos? Una medida de eficiencia es el tiempo que requiere un ordenador para resolver un problema usando un algoritmo para valores de entrada de un tamaño determinado. Una segunda medida es la cantidad de memoria necesaria para valores de entrada de un tamaño dado. Ambas preguntas están ligadas a la **complejidad computacional**.

Distinguimos entre **complejidad en tiempo**: número de operaciones que tiene que realizar el algoritmo; o **complejidad en espacio:** memoria necesaria. Esta última depende de la estructura de los datos usada en la implementación del algoritmo, por lo que escapa de nuestros objetivos. Nos limitaremos a estudiar la primera.

La **complejidad en tiempo** se describe en términos del número de operaciones requeridas, en lugar del tiempo de cálculo real, ya que distintos ordenadores necesitan tiempos diferentes para realizar la misma operación básica. Como operación básica para medir la complejidad en tiempo usaremos la comparación de enteros, la suma de enteros, la multiplicación de enteros, la división de enteros u otra operación básica.

Ejemplo 1: (Mayor elemento de un conjunto) Usamos como medida el número de comparaciones (operación básica usada).

```
In[]:= Conjunto={3, 6, 2, 1, 0, 7, 8, 5};
       Provisional=Conjunto[[1]];
       Do[
         If[Provisional<Conjunto[[i]],Provisional=Conjunto[[i]]];
       ,{i,2,Length[Conjunto]}];
       Print[“El mayor elemento es: ”, Provisional];
```

Fijado como máximo preliminar el elemento inicial de la lista, se realiza una comparación para determinar si se ha llegado al final de la lista y otra comparando el preliminar k y el segundo elemento de la lista, actualizando el valor del máximo si el segundo término es mayor que el primero. Como se hacen dos comparaciones desde el segundo hasta el n-ésimo, y una más para salir del bucle cuando i = n +1, se realizan exactamente $2(n – 1) + 1 = 2n – 1$ comparaciones.

Por tanto, este algoritmo es $\Theta(n)$, donde n es el número de elementos de “Conjunto”.

Ejemplo 2: (Búsqueda lineal) De nuevo, la operación básica usada es la comparación.

```
In[]:= Elemento=0;
       Conjunto={3, 6, 2, 1, 0, 7, 8, 5};
       i=1;
       While[Elemento!=Conjunto[[i]] && i!=Length[Conjunto],
             i++;
        ];
       If[Elemento==Conjunto[[i]],
          Print[“El elemento está en la posición: ”, i];
          ,Print[“El elemento no está”];
        ];
```

Ejemplo 2: (Búsqueda lineal)

En cada paso del bucle se llevan a cabo dos comparaciones: una para ver si se ha llegado al final de la lista y otra para comparar x y el elemento a_i. Finalmente, fuera del bucle se hace una comparación más. Por tanto, si $x = a_i$, se hacen $2i + 1$ comparaciones. El número máximo de comparaciones lo hacemos si x no está en la lista, $2n + 2$ ($2n$ para determinar que $x \neq a_i$ para todo i, una para salir del bucle y otra fuera del bucle).

Por tanto, una busqueda lineal en el peor de los caso tiene complejidad O(n).

El tipo de análisis que hemos realizado en el ejemplo anterior se denomina **peor caso**: el mayor número de operaciones que hace falta para resolver el problema dado usando el algoritmo para unos datos de entrada de un tamaño fijado.

Otro tipo importante de análisis de complejidad, es el llamado análisis del **caso promedio**. En este tipo se busca el número promedio de operaciones realizadas para resolver un problema considerando todas las posibles entradas de un tamaño fijo.

En el siguiente ejemplo, vamos a describir el análisis del caso promedio para el algoritmo de búsqueda lineal, suponiendo que el elemento x está en la lista.

Ejemplo 2: (Búsqueda lineal) Describimos ahora como se comporta este algoritmo suponiendo que x está en la lista: Si x está hay n posibles soluciones. Si x es el primer elemento, necesitamos tres comparaciones (una para comprobar si se ha alcanzado el final, otra para compara x con el elemento y otra fuera del bucle). Si x es el segundo se necesitan dos más y así sucesivamente. Por tanto el número promedio es:

$$\frac{3+5+7+...+(2n+1)}{n} = \frac{2(1+2+3+...+n)+n}{n} = \frac{2(\frac{n(n+1)}{2})+n}{n} = n+2$$

Por tanto, es $\Theta(n)$.

La terminología que comúnmente se usa para describir la complejidad de los algoritmos viene dada en la siguiente tabla:

Complejidad	Terminología
O(1)	Constante
O(log n)	Logarítmica
O(n)	Lineal
O(nlog n)	n log n
$O(n^k)$	Polinómica
$O(a^n)$, a > 1	Exponencial
O(n!)	Factorial

Un algoritmo que encuentra el mayor de una lista de 25 términos aplicando el algoritmo de busqueda anterior tiene **complejidad constante** pues usa 24 comparaciones. El algoritmo de busqueda lineal tiene **complejidad lineal** (en el peor de los casos y en el caso promedio).

Un problema que se puede resolver utilizando un algoritmo con complejidad polinómica en el peor caso se dice **tratable**, pues se espera que el algoritmo nos de la solución para una entrada de tamaño razonable en un tiempo relativamente corto. Sin embargo, si el polinomio de estimación en notación O tiene grado alto o los coeficiente son excesivamente grandes, el algoritmo puede necesitar demasiado tiempo para su utilización.

Diremos que un problema es **intratable** si no se puede resolver usando un algoritmo con complejidad polinómica en el peor caso. En la práctica, hay situaciones en las que un algoritmo puede ser capaz de resolver un problema mucho más rápido en la mayoría de los casos que en su peor caso. En tales casos, la complejidad media es una mejor medida del tiempo usado por un algoritmo para resolver un problema.

En la práctica, hay situaciones en las que un algoritmo puede resolver un problema mucho más rápido en la mayoría de los casos que en el peor caso. En tales casos la complejidad en tiempo en el caso promedio es una medida más acertada que el peor caso. Muchos problemas en la industria son intratables, pero pueden resolverse para todos los conjuntos de datos que aparecen en la vida real. Otra forma de manejar este tipo de problemas es buscar soluciones aproximadas del problema en lugar de las soluciones exactas.

Existe algunos problemas para los cuales incluso se puede probar que no existen algoritmos que puedan resolverlos, estos problemas se les llama **irresolubles** o **no resolubles**, en oposición a los problemas **resolubles**, para los que existen algoritmos que los resuelven.

Se dice que un problema pertenece a la **clase NP** si se puede comprobar en tiempo polinómico que una solución efectivamente lo es. Los problemas tratables se dicen que pertenecen a la **clase P**. Por último, una clase importante de problemas son los llamados **problemas NP-completos**, con la propiedad de que si alguno de estos problemas se puede resolver haciendo uso de un algoritmo con complejidad polinómica en el peor caso, entonces todos ellos se pueden resolver por medio de algoritmos con complejidad polinómica en el peor caso.

Terminamos describiendo la complejidad en tiempo de otro algoritmo:

Ejemplo 3: (Búsqueda binaria) Si suponemos que $n = 2^k$ ($k = \log_2 n$), en otro caso podemos considerar que la lista de elementos es parte de otra más larga con un número de elementos potencia de 2.

```
In[]:= Elemento=7;
       Conjunto={1,2,3,4,5,6,7,8,9,10};
       Principio=1;
       Final=Length[Conjunto];
       While[Principio≠Final,
             Mitad=Quotient[Principio+Final,2];
             If[Conjunto[[Mitad]]>=Elemento,
               Final=Mitad, Principio=Mitad+1
              ];
       ];
       If[Elemento==Conjunto[[Principio]],
          Print[“El elemento está en la posición: ”, Principio];,
          Print[“El elemento no está”];
       ];
```

Pasos:

(1) Principio=1, Final=10, Mitad=5
{1,2,3,4,5} {6,7,8,9,10}
(2) Principio=6, Final=10, Mitad=8
{6,7,8} {9,10}
(3) Principio=6, Final=8, Mitad=7
{6,7} {8}
(4) Principio=6, Final=7, Mitad=6
{6} {7}
(5) Principio=7, Final=7
Se compara “Elemento” y 7

Ejemplo 3: (Búsqueda binaria)

En cada paso del algoritmo, i y j, las posiciones del primero y último término de la lista restringida considerada se comparan para ver que la nueva lista tiene más de un término. Si $i \neq j$, se realiza una comparación para determinar si x es mayor que el término central de la lista restringida.

En el primer paso, la búsqueda se restringe a 2^{k-1} elementos (hasta aquí se han hecho dos comparaciones). Este proceso continua realizando dos comparaciones en cada paso. Finalmente, cuando queda un elemento (se ha realizado 2k comparaciones), una comparación nos dice que no hay más elementos, y una comparación más se utiliza para determinar si este término es x.

Por tanto, tenemos que realizar $2k + 2 = 2 \log n + 2$ comparaciones. Así, que una búsqueda binaria requiere como máximo $\Theta(\log n)$ comparaciones.

Otro tipo de algoritmos muy comunes son los dedicados a ordenar los elementos de una lista. Si tenemos una lista de elementos de un conjunto para los que conocemos una forma de ordenar. Una **ordenación** es colocar estos elementos en una lista en la cual los elementos se dispongan en orden creciente.

Ejemplo 4. Ordenación de burbuja, es uno de los más simple, pero no de los más eficientes. Damos sucesivas pasadas sobre la sucesión, intercambiando las posiciones de los elementos contiguos cuando no estén ordenados hasta que demos una pasada en la que no intercambiemos ninguna posición. Su complejidad es $\Theta(n^2)$.

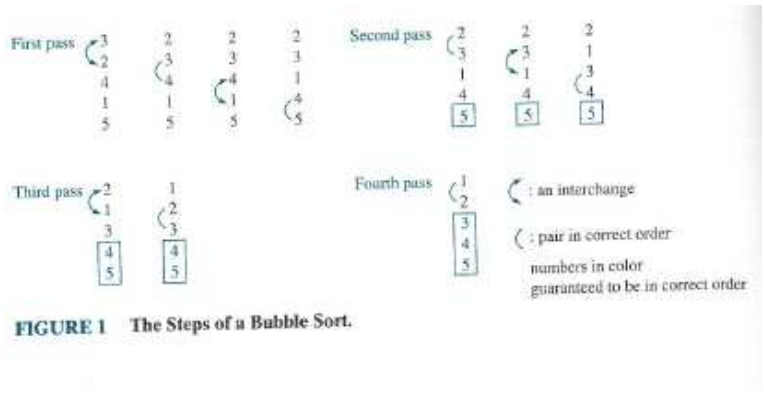

FIGURE 1 The Steps of a Bubble Sort.

Matemática Discreta
García Muñoz, M.A.

Ejemplo 5. Ordenación por inserción. Se añaden los elementos de uno en uno, colocándolos en su posición correcta (ordenados).

Ejemplo 6. Algoritmos voraces. Son algoritmos que hacen la que aparentemente es la mejor elección a cada paso. Por ejemplo, si queremos devolver el cambio exacto de cierta cantidad usando una determinada clase de monedas y siempre elegimos la moneda más grande que podamos a cada paso.